

PROGRAMACIÓN ORIENTADA A OBJETOS

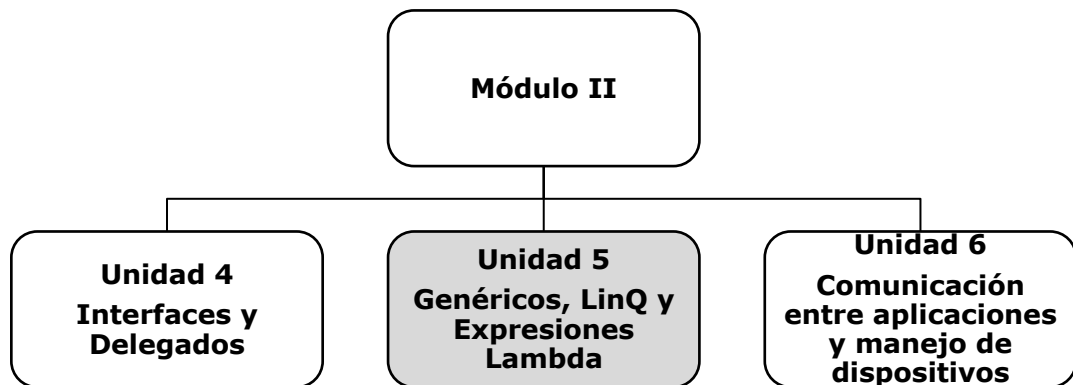
Módulo II

Programación de aplicaciones utilizando la técnica de la Programación Orientada a Objetos

Unidad 5

Genéricos, LinQ y Expresiones Lambda.

Docente titular y autor de contenidos: Prof. Ing. Darío Cardacci



Presentación

En esta unidad ...

Por todo lo expresado hasta aquí es que esperamos que usted, a través del estudio de esta unidad, se oriente hacia el logro de las siguientes metas de aprendizaje:

- Reconocer cuando es apropiado utilizar Genéricos para flexibilizar el código desarrollado.
- Contruir consultas con LinQ.
- Mejorar la escritura del código utilizando expresiones Lambda.

Los siguientes **contenidos** conforman el marco teórico y práctico que contribuirá a alcanzar las metas de aprendizaje propuestas:

Introducción a los genéricos. Ventaja de usar genéricos. Clases genéricas. Interfaces genéricas y métodos genéricos.

LINQ to Object. Introducción a las consultas con LINQ. Escritura de consultas con LINQ. Retorno y almacenamiento de consultas LINQ. Grupos anidados y subconsultas con LINQ.

Introducción a las expresiones lambda. Funciones anónimas. Uso de expresiones lambda en consultas.

A continuación, le presentamos un detalle de los contenidos y actividades que integran esta unidad. Usted deberá ir avanzando en el estudio y profundización de los diferentes temas, realizando las lecturas requeridas y elaborando las actividades propuestas, algunas de desarrollo individual y otras para resolver en colaboración con otros estudiantes y con su profesor tutor.

Contenidos y Actividades

1. Genéricos



Lectura requerida

- <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/>

2. LinQ

- [https://msdn.microsoft.com/es-es/library/bb397676\(v=vs.120\).aspx](https://msdn.microsoft.com/es-es/library/bb397676(v=vs.120).aspx)
- [https://msdn.microsoft.com/es-es/library/bb397900\(v=vs.120\).aspx](https://msdn.microsoft.com/es-es/library/bb397900(v=vs.120).aspx)
- [https://msdn.microsoft.com/es-es/library/bb397926\(v=vs.120\).aspx](https://msdn.microsoft.com/es-es/library/bb397926(v=vs.120).aspx)

3. Expresiones Lambda

- <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/anonymous-functions>
- <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions>

BIBLIOGRAFÍA OBLIGATORIA

Deitel Harvey M. Y Paul J. Deitel. **Cómo programar en C#**. Segunda edición. Pearson. México 2007

Lo/a invitamos ahora a comenzar con el estudio de los contenidos que conforman esta unidad.

1. Genéricos

A partir de la versión 2.0 del lenguaje C # se agregaron los **genéricos**. Los **genéricos** introducen en .NET el concepto de parámetros de tipo. Los parámetros de tipo permiten diseñar clases y métodos que difieren su implementación hasta que la clase se instancia o el método es utilizado por código de cliente.

Los **genéricos** se utilizan para maximizar la reutilización del código. En el namespace System.Collections.Generic de .NET, existen varias colecciones que utilizan **genéricos**. Se pueden crear interfaces **genéricas**, clases, métodos, eventos y delegados personalizados usando **genéricos**. La información sobre los tipos que se utilizan en un tipo de datos genérico se puede obtener en tiempo de ejecución mediante el uso de la reflexión.

Los **genéricos** posibilitan una solución a la limitación que implica tener que tipar los elementos a un nivel de abstracción alto, como puede ser object.

Veamos el costo operativo de toparnos con esta limitación.

```
1 referencia
private void Form1_Load(object sender, EventArgs e)
{
    System.Collections.ArrayList lista1 = new System.Collections.ArrayList();
    lista1.Add(3);
    lista1.Add(105);

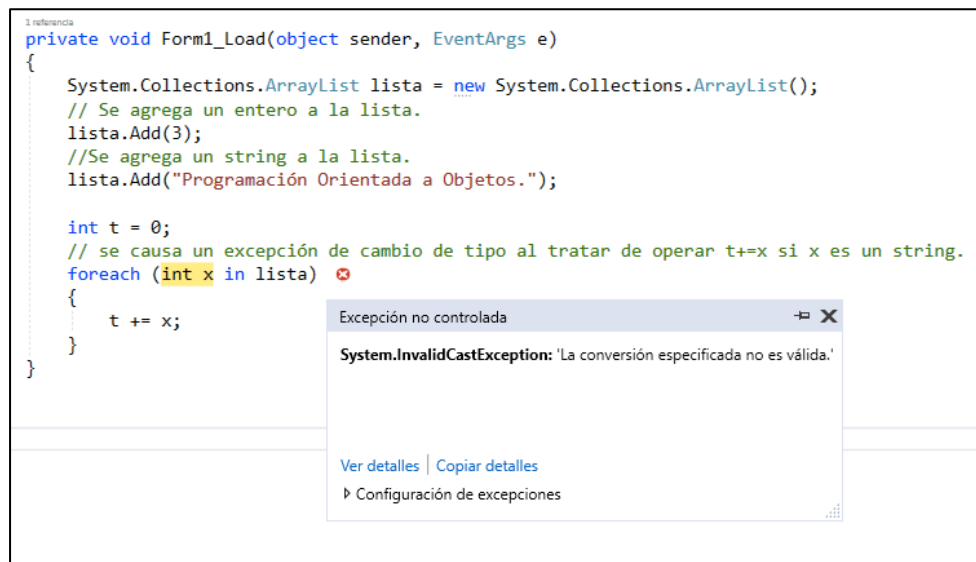
    System.Collections.ArrayList lista2 = new System.Collections.ArrayList();
    lista2.Add("Programación Orientada a Objetos.");
    lista2.Add("Programación Estructurada.");
}
```

EJ0001

El código del ejemplo **EJ0001** expone como se pueden generar dos **ArrayList** sin utilizar genéricos. En el primer caso se colocan enteros y en el segundo textos. En ambos casos, esa información internamente es tratada como **object**. Esto es debido a que al hacerlo así, nos independizamos del tipo de dato que se le desee incluir al **ArrayList**. No obstante, esto acarrea un alto costo, sobre todo si el dato que estamos trabajando corresponde al espacio de los **value type**. El costo de

procesamiento está dado por la necesidad de pasar un entero (**int** – **value type**) a un **object** que es un **reference type**. Para que sea posible se debe proceder a realizar un **boxing** (proceso por el cual un value type se empaqueta como reference Type, para ser tratado dentro de la memoria administrada o heap). También, al consumir ese valor de la lista se deberá incurrir en una operación de **unboxing** (operación que desempaqueta el reference type para que pueda ser consumido en su tipo original (value type)). Las operaciones de **boxing** y **unboxing**, si bien son necesarias en algunos contextos, siempre que se puedan evitar, es conveniente evitarlas.

Otro problema aparejado a utilizar estas formas de trabajo, son las relacionadas con el control de tipos. En el ejemplo **EJ0002** observaremos como se produce un error en tiempo de ejecución.



```
1 referencia
private void Form1_Load(object sender, EventArgs e)
{
    System.Collections.ArrayList lista = new System.Collections.ArrayList();
    // Se agrega un entero a la lista.
    lista.Add(3);
    //Se agrega un string a la lista.
    lista.Add("Programación Orientada a Objetos.");

    int t = 0;
    // se causa un excepción de cambio de tipo al tratar de operar t+=x si x es un string.
    foreach (int x in lista)
    {
        t += x;
    }
}
```

Excepción no controlada

System.InvalidCastException: 'La conversión especificada no es válida.'

[Ver detalles](#) | [Copiar detalles](#)

► Configuración de excepciones

EJ0002

La lista, al estar compuesta por enteros y textos lo cual en principio parece una ventaja, ya que internamente todos son tratados como object, termina generando un error en tiempo de ejecución al querer acumular en la variable **t** de tipo **int** un texto.

A continuación, se verá como utilizando genéricos se logra el control de tipos en tiempo de compilación (antes de ejecutar). El ejemplo **EJ0003**, da muestra de ello.

```
private void Form1_Load(object sender, EventArgs e)
{
    // lista genérica donde se le hace explícito que tipo de dato contendrá
    // colocando List<int>
    List<int> lista = new List<int>();

    // No se procede a la operación de Boxing, debido a que internamente la lista administra int
    lista.Add(3);

    // Error en tiempo de compilación que indica que no se puede ingresar un string a la lista de enteros
    lista.Add("Programación-Orientada a Objetos.");
}
```

class System.String
Represents text as a sequence of UTF-16 code units. To browse the .NET Framework source code for this type, see the Reference Source.
Argumento 1: no se puede convertir de 'string' a 'int'

EJ0003

Las clases genéricas encapsulan operaciones que no son específicas de un tipo de datos en particular. Esto trae aparejada la necesidad de poder indicarle a la operación, dinámicamente, que tipo debe adoptar en cada caso particular para su funcionamiento.

El uso más común para las clases genéricas se observa en colecciones como listas vinculadas, tablas hash, stacks, colas, árboles, etc. Las operaciones como agregar y eliminar elementos de la colección, se realizan básicamente de la misma manera, independientemente del tipo de dato almacenado.

Cuando se crean clases genéricas, se deben considerar aspectos importantes. A continuación, se mencionan dos muy trascendentes:

- Qué tipos generalizar en parámetros de tipo.

Como regla general, cuantos más tipos pueda parametrizar, más flexible y reutilizable será su código. Sin embargo, demasiada flexibilización puede crear código que es difícil de leer o comprender para otros desarrolladores.

- ¿Qué restricciones, si las hay, se deben aplicar a los parámetros de tipo? (El tema restricciones de parámetros de tipo se analizará más adelante en este texto)

Una buena regla es aplicar las máximas restricciones posibles. Esto es debido a que ayudan a limitar que tipos se pueden enviar como parámetros de tipo.

Por ejemplo, si sabe que la clase genérica está destinada a ser utilizada solo con tipos de referencia, se aplica una restricción de clase. Eso evitará el uso involuntario de tipos de valores.

Otro ejemplo, podría ser si la clase genérica trabajará con subclases derivadas de **Empleado**. En este caso se aplica una restricción de clase de tipo **Empleado**.

Los parámetros de tipo y las restricciones poseen reglas que tienen varias implicaciones para el comportamiento de la clase genérica, especialmente con respecto a la herencia y la accesibilidad de los miembros.

Las clases genéricas pueden heredar de clases base: concretas, genéricas construidas cerradas o genéricas construidas abiertas. El ejemplo **EJ0004** muestra lo expresado.

```
class ClaseNodo { } // Clase concreta
// 2 referencias

class ClaseNodoGenerica<T> { } // Clase genérica

// Clase genérica que hereda de una clase concreta
// 0 referencias
class NodoConcreto<T> : ClaseNodo { }

// Clase genérica que hereda de una clase genérica cerrada(pues al parámetro de tipo se le indica que es int)
// 0 referencias
class NodoCerrado<T> : ClaseNodoGenerica<int> { }

// Clase genérica que hereda de una clase genérica abierta.
// Si se le asigna un tipo a NodoAbierto, el mismo será utilizado por ClaseNodoGenerica.
// 0 referencias
class NodoAbierto<T> : ClaseNodoGenerica<T> { }
```

EJ0004

Las clases no genéricas, en otras palabras, concretas, pueden heredar de las clases base cerradas, pero no de las clases genéricas abiertas. Esto es debido a que no hay forma que el código cliente proporcione el argumento de tipo válido, el cual es requerido para crear una instancia de la clase base. En el ejemplo **EJ0005** se puede observar como **NodoA** hereda correctamente mientras que **NodoB** genera un error al no poder determinar un tipo para **T**.

```
class ClaseBaseGenerica<T> { }
// 2 referencias

// Correcto
// 0 referencias
class NodoA : ClaseBaseGenerica<int> { }

// Genera un error donde indica que T no se encontró.
// 0 referencias
class NodoB : ClaseBaseGenerica<T> { }

// Generates an error
// class NodoC : T { }
```

El nombre del tipo o del espacio de nombres 'T' no se encontró (¿falta una directiva using o una referencia de ensamblado?)
Mostrar posibles correcciones (Alt+Entrar o Ctrl+.)

EJ0005

A continuación, en el ejemplo **EJ0006** se analiza una clase con un parámetro de tipo. el tipo enviado al parámetro de tipo **<T>** se utiliza dentro de la clase para la definición de un campo, el parámetro del constructor, el parámetro de un método y una propiedad.

```
class ClaseGenerica<T>
{
    private T CampoGenerico;

    2 referencias
    public ClaseGenerica(T value)
    {
        CampoGenerico = value;
    }

    2 referencias
    public void MetodoGenerico(T pGenerico)
    {
        MessageBox.Show("El constructor recibió el valor: " + CampoGenerico);
        MessageBox.Show("Tipo de Parámetro: " + typeof(T).ToString() + " " + pGenerico);
    }

    4 referencias
    public T PropiedadGenerica { get; set; }
}
```

EJ0006

Como se observa los miembros de la clase **ClaseGenerica** se tipan con el tipo que recibe **<T>**. Esto permite que las instancias de **ClaseGenerica** puedan funcionar con diversos tipos.

En este ejemplo se utilizan dos tipos distintos, el **int** y el **string**, como se muestra a continuación:

```
private void button1_Click(object sender, EventArgs e)
{
    ClaseGenerica<int> G = new ClaseGenerica<int>(4);
    G.MetodoGenerico(10);
    G.PropiedadGenerica = 20;
    MessageBox.Show("La propiedad posee el valor: " + G.PropiedadGenerica);
}
```

EJ0006


```
private void button2_Click(object sender, EventArgs e)
{
    ClaseGenerica<string> G = new ClaseGenerica<string>("Valor 1");
    G.MetodoGenerico("Valor 2");
    G.PropiedadGenerica = "Valor 3";
    MessageBox.Show("La propiedad posee el valor: " + G.PropiedadGenerica);
}
```

EJ0006

Cuando utilizamos parámetros de tipo se puede restringir que tipo se puede enviar al parámetro.

Las restricciones le informan al compilador sobre las capacidades que debe tener un argumento de tipo. Sin ninguna restricción, el argumento de tipo podría ser de cualquier tipo. Si el código del cliente intenta crear una instancia de su clase utilizando un tipo que no está permitido por una restricción, el resultado es un error en tiempo de compilación. Las restricciones se especifican utilizando la palabra clave **where**. La siguiente tabla enumera los siete tipos de restricciones:

Sintaxis	Descripción
where T : struct	El argumento de tipo debe ser un tipo de valor. Cualquier tipo de valor excepto Nullable puede especificarse.
where T : class	El argumento de tipo debe ser un tipo de referencia. Esta restricción se aplica también a cualquier clase, interfaz, delegado o tipo de matriz.
where T : unmanaged	El argumento de tipo no debe ser un tipo de referencia y no debe contener ningún miembro de tipo de referencia en ningún nivel de anidamiento.
where T : new()	El argumento de tipo debe tener un constructor sin parámetros público. Cuando se usa conjuntamente con otras restricciones, la restricción <code>new()</code> debe especificarse en último lugar.
where T : <nombre de la clase base>	El argumento de tipo debe ser o derivarse de la clase base especificada.
where T : <nombre de la interfaz>	El argumento de tipo debe ser o implementar la interfaz especificada. Pueden especificarse varias restricciones de interfaz. La interfaz de restricciones también puede ser genérica.
where T : U	El argumento de tipo proporcionado por T debe ser o derivarse del argumento proporcionado para U.

En siguiente ejemplo muestra como generar una clase, con un parámetro de tipo que posee una restricción. La restricción establece que solo se

podrá usar como argumento un tipo **Empleado** u otro derivado de **Empleado**.

Suponiendo que tenemos las siguientes clases:

```
public class Empleado
{
    3 referencias
    public string Nombre { get; set; }
    3 referencias
    public string Apellido { get; set; }
}
4 referencias
public class EmpleadoAdministrativo : Empleado
{
    1 referencia
    { public EmpleadoAdministrativo(string pNombre, string pApellido) { Nombre = pNombre; Apellido = pApellido; } }
4 referencias
public class EmpleadoVenta : Empleado
{
    1 referencia
    { public EmpleadoVenta(string pNombre, string pApellido) { Nombre = pNombre; Apellido = pApellido; } }
0 referencias
public class Socio
{ }
```

EJ0007

Además, tenemos la clase **MuestraEmpleado** que es una clase con un parámetro de tipo.

```
public class MuestraEmpleado<T> where T : Empleado
{
    2 referencias
    public string Mostrar(T pEmpleado)
    {
        return pEmpleado.Nombre + " " + pEmpleado.Apellido;
    }
}
```

EJ0007

Al utilizarla, se le puede pasar al parámetro **T** un tipo **Empleado**, **EmpleadoAdministrativo** o **EmpleadoVenta**.

Si se le pasa otro tipo generará un error en tiempo de compilación. El siguiente fragmento de código muestra como se puede utilizar correctamente.

```
private void button1_Click(object sender, EventArgs e)
{
    MuestraEmpleado<EmpleadoAdministrativo> ME = new MuestraEmpleado<EmpleadoAdministrativo>();
    MessageBox.Show(ME.Mostrar(new EmpleadoAdministrativo("Ana", "Martinez")));
}

1 referencia
private void button2_Click(object sender, EventArgs e)
{
    MuestraEmpleado<EmpleadoVenta> ME = new MuestraEmpleado<EmpleadoVenta>();
    MessageBox.Show(ME.Mostrar(new EmpleadoVenta("Juan", "Perez")));
}
```

EJ0007

Se puede observar que tipo de error se genera si le pasamos un tipo distinto a **Empleado**.

```
private void button3_Click(object sender, EventArgs e)
{
    // Genera un error ya que el tipo socio no es un Empleado.
    MuestraEmpleado<Socio> ME = new MuestraEmpleado<Socio>();
    //MessageBox.Sh...
}

class EJ0007.Socio
El tipo 'EJ0007.Socio' no se puede usar como parámetro de tipo 'T' en el tipo o método genérico 'MuestraEmpleado<T>'. No hay ninguna conversión de referencia implícita de 'EJ0007.Socio' a 'EJ0007.Empleado'.
```

EJ0007

Los parámetros de tipo también se pueden aplicar a los métodos de una clase. Estos métodos se denominan métodos genéricos. En el ejemplo **EJ0008** se ejemplifica como desarrollarlo.

```
public class Intercambio
{
    2 referencias
    public void Ejecutar<T>(ref T valor1, ref T valor2)
    {
        T aux = valor1;
        valor1 = valor2;
        valor2 = aux;
    }
}
```

EJ0008

En este ejemplo la clase **Intercambio** realiza un swap entre dos valores. Como se puede observar el método **Ejecutar** recibe un tipo en el parámetro de tipo **<T>**. El tipo recibido se utiliza para tipar los

parametros **valor1** y **valor2**. Al utilizar el método se le pueden pasar valores de distintos tipos como se observa a continuación.

```
1 referencia
private void button1_Click(object sender, EventArgs e)
{
    var I = new Intercambio();
    var Valor1 = int.Parse(Interaction.InputBox("Valor 1: "));
    var Valor2 = int.Parse(Interaction.InputBox("Valor 2: "));
    MessageBox.Show("Antes de Intercambiar - Valor 1: " + Valor1.ToString() + " Valor 2: " + Valor2.ToString());
    I.Ejecutar<int>(ref Valor1, ref Valor2);
    MessageBox.Show("Después de Intercambiar - Valor 1: " + Valor1.ToString() + " Valor 2: " + Valor2.ToString());

    var Cadena1 = Interaction.InputBox("Cadena 1: ");
    var Cadena2 = Interaction.InputBox("Cadena 2: ");
    MessageBox.Show("Antes de Intercambiar - Cadena 1: " + Cadena1 + " Cadena 2: " + Cadena2);
    I.Ejecutar<string>(ref Cadena1, ref Cadena2);
    MessageBox.Show("Después de Intercambiar - Cadena 1: " + Cadena1 + " Cadena 2: " + Cadena2);
}
```

EJ0008

2. LinQ

LinQ o Language Integrated Query es un conjunto de herramientas para realizar consultas a distintas fuentes de datos: objetos, xmls, bases de datos, etc... Para ello, usa un tipo de funciones propias, que unifica las operaciones más comunes de consulta de datos en todos los entornos. Con esto, se consigue un lenguaje para todas las tareas relacionadas con consultas de datos.

La sintaxis es parecida a la existente en SQL, pero con la ventaja de que tenemos toda la potencia de .net y visual studio a la hora de codificar.

Las tres partes básicas de una expresión de consulta **LinQ** son:

1. Obtener el origen de datos.
2. Crear la consulta.
3. Ejecutar la consulta. ←

En el ejemplo **EJ0009** se puede observar esto con claridad.

Las consultas LINQ siempre se ejecutan cuando se recorre en iteración la variable de consulta, no cuando se crea la citada variable de consulta. Esto se denomina ejecución aplazada.

```

IEnumerable<int> Vquery;
private int[] numeros;
1 referencia
public EjemploLINQ(int[] pDatos)
{
    // Las tres partes de una consulta LinQ:
    // 1. Origen de Datos.
    numeros = pDatos;
    // 2. Creación de la consulta.
    Vquery =
        from num in numeros
        where (num % 2) == 0
        select num;
}
1 referencia
public string[] RetornaPares()
{
    // 3. Ejecución de la consulta.
    string numerosRetorno="";
    foreach (int num in Vquery)
    {
        numerosRetorno += num + ",";
    }
    numerosRetorno = numerosRetorno.Substring(0, numerosRetorno.Length - 1);
    char[] S = { ',' };
    return numerosRetorno.Split(S);
}

```

EJ0009

En este ejemplo el origen de datos lo constituye un array de números enteros. Se almacena en el campo privado denominado **numeros**. Esto es así pues el array es compatible con **IEnumerable<T>**. El origen de datos podrá ser **IEnumerable<T>** o algún tipo derivado.

La consulta, especifica qué información se recuperará del origen de datos. Opcionalmente, una consulta también especifica cómo se debe ordenar, agrupar y configurar esa información antes de devolverla. Una consulta se almacena en una variable de consulta, en nuestro ejemplo en una variable de tipo **IEnumerable<int>** denominada **Vquery**.

La consulta en el ejemplo anterior devuelve todos los números pares del array utilizado como origen de datos. La expresión de consulta contiene tres cláusulas: **from**, **where** y **select**.

La cláusula **from** especifica la fuente de datos, la cláusula **where** aplica el filtro y la cláusula **select** especifica el tipo de los elementos devueltos.

La ejecución de la consulta se produce cuando se ejecuta el método **RetornaPares**. El mismo consume la consulta almacenada en la variable **Vquery** utilizando un **for..each**.

La utilización de la clase **EjemploLINQ** que contiene el código LinQ se consume desde:

```
private void button1_Click(object sender, EventArgs e)
{
    int[] Vnumeros = new int[7] { 1, 2, 3, 4, 5, 6, 7 };
    var ELQ = new EjemploLINQ(Vnumeros);
    string[] Vpares = ELQ.RetornaPares();
    this.listBox1.Items.Clear(); this.listBox1.Items.AddRange(Vpares);
}
```

EJ0009

Adaptemos el ejemplo para conocer la cantidad de números pares que posee el array. Esto puede observarse en el ejemplo **EJ0010**.

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show(new EjemploLINQ().
        RetornaPares(new int[] { 1, 2, 3, 4, 5, 6, 7 }).ToString() + " números pares !!!");
}

1 referencia
class EjemploLINQ
{
    1 referencia
    public int RetornaPares(int[] pDatos)
    { return (from num in pDatos where (num % 2) == 0 select num).Count<int>();}
}
```

EJ0010

Este ejemplo utiliza **Count** sobre el resultado de la consulta para obtener la cantidad de elementos. El retorno de la función **RetornaPares** es de tipo **int**.

Si deseamos que retorne una lista podemos desarrollar lo que expone el ejemplo **EJ0011**. En este caso utilizamos **ToList<int>**. El retorno de la función **RetornaPares** es de tipo **List<int>**.

Para consumir la expresión lambda, se utiliza a lista retornada por:

```
new EjemploLINQ().RetornaPares(new int[] { 1, 2, 3, 4, 5, 6, 7 })
```

Luego a esa lista se le aplica un:

```
ForEach(X => S += X + " ")
```

Con esto se logra recorrer cada elemento de la lista retornada y por cada elemento **"X"** recorrido, el mismo se concatena al string **"S"**

```

private void button1_Click(object sender, EventArgs e)
{
    var S = "";
    new EjemploLINQ().RetornaPares(new int[] { 1, 2, 3, 4, 5, 6, 7 }).ForEach(X => S += X + " ");
    MessageBox.Show(S + "son números pares !!!");
}
}
1 referencia
class EjemploLINQ
{
    1 referencia
    public List<int> RetornaPares(int[] pDatos)
    { return (from num in pDatos where (num % 2) == 0 select num).ToList<int>(); }
}

```

EJ0011

Avancemos incorporando algunas operaciones básicas que potenciarán lo que hacemos con **LinQ**.

Para realizar esto se utilizarán las siguientes clases:

```

public class Cliente
{
    5 referencias
    public Cliente(string pNombre, string pApellido, string pTipo)
    {
        Nombre = pNombre;Apellido = pApellido;Tipo = pTipo;
    }
    3 referencias
    public string Nombre { get; set; }
    3 referencias
    public string Apellido { get; set; }
    6 referencias
    public string Tipo { get; set; }
    6 referencias
    public override string ToString()
    {
        return Nombre + " " + Apellido + " " + Tipo;
    }
}
6 referencias
public class Distribuidor
{
    2 referencias
    public Distribuidor(string pNombre, string pTipo) { Nombre = pNombre;Tipo = pTipo; }
    2 referencias
    public string Nombre { get; set; }
    2 referencias
    public string Tipo { get; set; }
}
}

```

EJ0012

También las siguientes listas con datos:

```

private void Form1_Load(object sender, EventArgs e)
{
    C.AddRange(new Cliente[] {new Cliente("Sol", "Fernandez", "Internacional"),
                               new Cliente("Juan", "Perez", "Nacional"),
                               new Cliente("Ariel", "Garcia", "Nacional"),
                               new Cliente("Cecilia", "Costa", "Internacional"),
                               new Cliente("Ana", "Martinez", "Nacional"),});

    D.AddRange(new Distribuidor[] { new Distribuidor("Distribuidora Tex", "Nacional"),
                                     new Distribuidor("Distribuidora InterTex", "Internacional")});
}

```

EJ0012

Ver todos los datos de Clientes

```

private void button1_Click(object sender, EventArgs e)
{
    var T = from cli in C select cli;
    foreach (Cliente Z in T.ToList<Cliente>())
    {
        this.listBox1.Items.Add(Z.ToString());
    }
}

```

EJ0012

Filtrar.

```

private void button2_Click(object sender, EventArgs e)
{
    var T = from cli in C where cli.Tipo=="Nacional" select cli;
    foreach (Cliente Z in T.ToList<Cliente>())
    {
        this.listBox2.Items.Add(Z.ToString());
    }
}

```

EJ0012

Filtrar con el operador lógico and (&&).

```
private void button3_Click(object sender, EventArgs e)
{
    var T = from cli in C where cli.Tipo == "Nacional" && cli.Nombre[0]=='A' select cli;
    foreach (Cliente Z in T.ToList<Cliente>())
    {
        this.listBox3.Items.Add(Z.ToString());
    }
}
```

EJ0012

Ordenamiento.

```
private void button4_Click(object sender, EventArgs e)
{
    var T = from cli in C orderby cli.Apellido ascending select cli;
    foreach (Cliente Z in T.ToList<Cliente>())
    {
        this.listBox4.Items.Add(Z.ToString());
    }
}
```

EJ0012

Agrupamiento.

```
private void button5_Click(object sender, EventArgs e)
{
    var T = from cli in C group cli by cli.Tipo;

    // customerGroup is an IGrouping<string, Customer>
    foreach (var GrupoClientes in T)
    {
        this.listBox5.Items.Add(GrupoClientes.Key);
        foreach (var Cliente in GrupoClientes)
        {
            this.listBox5.Items.Add("    " + Cliente.ToString());
        }
    }
}
```

EJ0012

Unión.

```
private void button6_Click(object sender, EventArgs e)
{
    var JoinCD = from cli in C join dis in D on cli.Tipo equals dis.Tipo
                 select new { Cliente = cli.ToString(), Distribuidor = dis.Nombre };
    foreach (var Z in JoinCD.ToList())
    {
        this.listBox6.Items.Add(Z.Cliente + " " + Z.Distribuidor);
    }
}
```

EJ0012

También se puede utilizar LinQ para transformar datos de distintos orígenes de datos. Al usar una consulta LINQ, puede usar una secuencia como entrada y modificarla de muchas maneras, para crear una nueva secuencia de salida. Pero quizás la característica más poderosa de las consultas LINQ es la capacidad de crear nuevos tipos. Esto se logra con la cláusula **select**. Por ejemplo, se puede realizar las siguientes tareas:

- Combinar múltiples secuencias de entrada en una sola secuencia de salida que tiene un nuevo tipo.
- Crea secuencias de salida cuyos elementos consisten en solo una o varias propiedades de cada elemento en la secuencia de origen.
- Crea secuencias de salida cuyos elementos consisten en los resultados de las operaciones realizadas en los datos fuente.
- Crea secuencias de salida en un formato diferente. Por ejemplo, puede transformar datos de filas de SQL o archivos de texto en XML.

En el ejemplo **EJ0013** se analiza lo expresado. En el mismo existen Clientes y Proveedores. Cada uno en listas distintas y ambos poseen **Nombre**. Se desea consumir los nombres de los dos orígenes de datos y mostrarlos.

```

public class Cliente
{
    5 referencias
    public Cliente(string pNombre, string pApellido, string pLocalidad)
    {
        Nombre = pNombre; Apellido = pApellido; Localidad = pLocalidad;
    }
    2 referencias
    public string Nombre { get; set; }
    2 referencias
    public string Apellido { get; set; }
    2 referencias
    public string Localidad { get; set; }
    3 referencias
    public override string ToString()
    {
        return Nombre + " " + Apellido;
    }
}
6 referencias
public class Proveedor
{
    2 referencias
    public Proveedor(string pNombre, string pLocalidad) { Nombre = pNombre; Localidad = pLocalidad; }
    2 referencias
    public string Nombre { get; set; }
    2 referencias
    public string Localidad { get; set; }
}

```

EJ0013

Los datos de cada lista son:

```

C.AddRange(new Cliente[] {new Cliente("Sol", "Fernandez", "Colegiales"),
                           new Cliente("Juan", "Perez", "Saavedra"),
                           new Cliente("Ariel", "Garcia", "Colegiales"),
                           new Cliente("Cecilia", "Costa", "Nuñez"),
                           new Cliente("Ana", "Martinez", "Belgrano"),});

P.AddRange(new Proveedor[] { new Proveedor("Distribuidora Tex", "Saavedra"),
                              new Proveedor("Distribuidora InterTex", "Nuñez")});

```

EJ0013

La consulta de **LinQ** que logra el objetivo es:

```

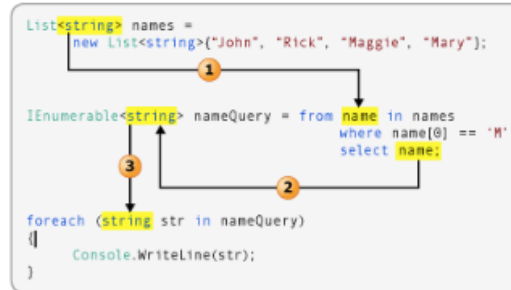
private void button1_Click(object sender, EventArgs e)
{
    var Resultado = (from cli in C where cli.Localidad == "Saavedra" select cli.ToString())
        .Concat(from pro in P where pro.Localidad == "Saavedra" select pro.Nombre);
    foreach (var p in Resultado)
    {
        this.listBox1.Items.Add(p);
    }
}

```

EJ0013

Consultas que no transforman los datos de origen.

La siguiente ilustración muestra una operación de consulta **LINQ to Objects** que no realiza transformaciones en los datos. La fuente contiene una secuencia de cadenas y la salida de consulta también es una secuencia de cadenas.

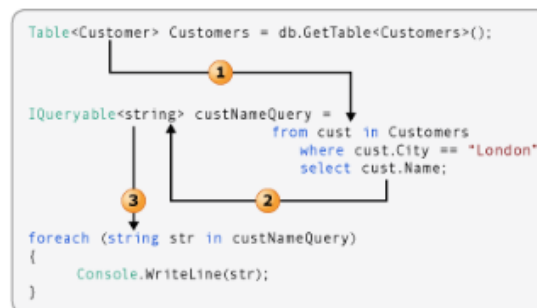


```
private void button1_Click(object sender, EventArgs e)  
{  
    List<string> nombres = new List<string> { "Juan", "María", "Mariana", "Pedro" };  
    IEnumerable<string> Z = from N in nombres where N[0] == 'M' select N;  
    foreach (string S in Z)  
    {  
        this.listBox1.Items.Add(S);  
    }  
}
```

EJ0014

Consultas que transforman los datos de origen.

La siguiente ilustración muestra una operación de consulta **LINQ to SQL** que realiza una transformación simple en los datos. La consulta toma una secuencia de **Cientes** como entrada y selecciona solo la propiedad **Nombre**. Como **Nombre** es un **string**, la consulta produce una secuencia de **string** como salida.



```

private void button1_Click(object sender, EventArgs e)
{
    List<Cliente> clientes = new List<Cliente> { new Cliente("Juan", "Londres"),
                                                new Cliente("María", "Madrid"),
                                                new Cliente("Mariana", "Buenos Aires"),
                                                new Cliente("Pedro", "Buenos Aires") };

    IEnumerable<string> Z = from C in clientes where C.Ciudad == "Buenos Aires" select C.Nombre ;
    foreach (string S in Z)
    {
        this.listBox1.Items.Add(S);
    }
}
7 referencias
public class Cliente
{
    4 referencias
    public Cliente(string pNombre, string pCiudad) { Nombre = pNombre; Ciudad = pCiudad; }
    2 referencias
    public string Nombre { get; set; }
    2 referencias
    public string Ciudad { get; set; }
}

```

EJ0015

3. Expresiones Lambda

Una **expresión lambda** es una función anónima que se puede usar para crear delegados. Al usar **expresiones lambda**, puede escribir funciones locales que se pueden pasar como argumentos, o devolver como el valor de las llamadas a funciones. Las **expresiones Lambda** son particularmente útiles para escribir expresiones de consulta LINQ.

Para crear una **expresión lambda**, especifique los parámetros de entrada (si corresponde) en el lado izquierdo del operador lambda \Rightarrow , y coloque la **expresión o el bloque de instrucciones** en el otro lado. Por ejemplo, la **expresión lambda** $X \Rightarrow X * X$, especifica un parámetro que se llama X y devuelve el valor de X al cuadrado. Puede asignar esta expresión a un tipo de delegado, como se muestra en el siguiente ejemplo:

```

delegate int del(int i);
1 referencia
private void button1_Click(object sender, EventArgs e)
{
    del myDelegate = x => x * x;
    MessageBox.Show(myDelegate(5).ToString());
}

```

EJ0016

El delegado se puede instanciar como **Func<int,bool> miFunc** donde **int** es un parámetro de entrada y **bool** es el valor de retorno. El valor de retorno siempre se especifica en último lugar, o sea como último parámetro de tipo. **Func<int, string, bool>** define un delegado con dos parámetros de entrada, **int** y **string**, y un tipo de retorno de **bool**. En el siguiente ejemplo **EJ0017**, el delegado **Func**, cuando se invoca, devolverá verdadero o falso para indicar si el parámetro de entrada es igual al valor a comparar:

```
private void button1_Click(object sender, EventArgs e)
{
    Func<int, bool> myFunc = x => x == int.Parse(this.textBox1.Text);
    bool result = myFunc(int.Parse(this.textBox2.Text)); // retorna verdadero o falso
    MessageBox.Show(result.ToString());
}
```

EJ0017

También puede suministrar una expresión lambda cuando el tipo de argumento es un **Expression<Func>**, por ejemplo, en los operadores de consulta estándar definidos en **System.Linq.Queryable**. El ejemplo **EJ0018** expone como se aplica esto en el método **Count** para contar los números impares.

```
private void button1_Click(object sender, EventArgs e)
{
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    MessageBox.Show(numbers.Count(n => n % 2 == 1).ToString());
}
```

EJ0018

En este ejemplo el compilador puede inferir el tipo del parámetro de entrada, también se puede especificar explícitamente.

El ejemplo **EJ0019** produce una secuencia que contiene todos los elementos de un vector numérico que están en el lado izquierdo del primer valor que es igual o mayor a 7, porque ese es el primer número en la secuencia que no cumple con la condición.

```
private void button1_Click(object sender, EventArgs e)
{
    int[] numeros = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    var numerosMenores7 = numeros.TakeWhile(n => n < 7);
    foreach (int n in numerosMenores7)
    {
        this.listBox1.Items.Add(n);
    }
}
```

EJ0019

Se pueden utilizar expresiones Lambda en consultas. El ejemplo **EJ0020** se muestra cómo usar una expresión lambda en una consulta, utilizando el operador de consulta estándar **Enumerable.Where**. Tenga en cuenta que el método **Where** en este ejemplo, tiene un parámetro de entrada del tipo de delegado Func <TResult> y que el delegado toma un entero como entrada y devuelve un booleano. La **expresión lambda** se puede convertir a ese delegado.

Enumerable.Where, filtra una secuencia de valores basada en un predicado.

Where posee la siguiente firma:

Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)

En este caso se filtran los números que sean menores o iguales al índice que ocupa en el array multiplicado por 10.

```
1 referencia
private void button1_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();
    int[] numbers = { 0, 30, 20, 15, 90, 85, 40, 75 };
    IEnumerable<int> query =
        numbers.Where((number, index) => number <= index * 10);

    foreach (int number in query)
    {
        listBox1.Items.Add(number);
    }
}
```

EJ0020

El ejemplo **EJ0021** muestra otra forma de aprovechar el **Where**.

```
1 referencia
private void button1_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();
    List<string> frutas =
    new List<string> { "manzana", "frutilla", "banana", "mango",
        "naranja", "uva", "pera", "durazno" };
    IEnumerable<string> consulta = frutas.Where(fruit => fruit.Length < 6);
    foreach (string fruta in consulta)
    {
        listBox1.Items.Add(fruta);
    }
}
```

EJ0021

El ejemplo **EJ0022** muestra como escribir el código del ejemplo **EJ0021** de manera más reducida.

```
1 referencia
private void button1_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();
    List<string> frutas =
    new List<string> { "manzana", "frutilla", "banana", "mango",
                     "naranja", "uva", "pera", "durazno" };

    foreach (string fruta in frutas.Where(fruit => fruit.Length < 6))
    {
        listBox1.Items.Add(fruta);
    }
}
```

EJ0022



Guía de Revisión Conceptual

1. ¿Qué son y para qué se usan los tipos genéricos?
2. ¿A partir de que versión de c# se pueden utilizar?
3. Enumere las ventajas de utilizar genéricos.
4. ¿A qué elementos se le pueden aplicar genericos?
5. ¿Cuáles son los usos más comunes de los genéricos?
6. ¿Qué aspectos trascendentes se deben considerar al crear clases genéricas?
7. ¿Cuál es la diferencia entre heredar de una clase genérica abierta y una clase genérica cerrada?
8. ¿Por qué es importante colocar restricciones en los parámetros de tipo?
9. ¿Cuáles son los tipos de restricciones que se le pueden colocar a un parámetro de tipo?
10. ¿Ejemplifique cómo crearía un método genérico con un parámetro de tipo?
11. ¿Qué es LinQ?
12. ¿Qué orígenes de datos se pueden consultar con LinQ?
13. ¿Cuáles son las partes básicas de una consulta LinQ?
14. ¿Qué especifica la consulta en una estructura de LinQ?
15. Mencione al menos tres cláusulas (las más importantes) que se usan en una expresión de consulta LinQ.
16. Explique que hace cada cláusula enumerada en la pregunta anterior
17. Dado que una expresión de consulta genera un IEnumerable, enumere y explique los métodos de extensión que posee IEnumerable. (p.e Count)
18. ¿Qué utilizaría para ordenar en una expresión LinQ?
19. ¿Qué utilizaría para lograr una unión entre dos orígenes de datos en una expresión LinQ?
20. ¿Cómo se pueden generar nuevos tipos utilizando LinQ?
21. ¿Qué es una expresión Lambda?

22. Ejemplifique cómo se puede utilizar una expresión lambda para realizar una consulta.
23. ¿Qué debo realizar para crear una expresión lambda?
24. ¿Indique como puede declarar un delegado utilizando Func y que significa cada elemento utilizado?
25. Crear un delegado utilizando Func que posea tres parámetros de entrada y uno de salida. Los parámetros de entrada son: el primero **int**, el segundo **double** y el tercero **bool**. El parámetro de retorno es de tipo **bool**.

Cierre de la unidad

- Se recomienda que el alumno realice una lectura detallada de los contenidos señalados.
- Es importante que vea los ejemplos propuestos e intente contestar las preguntas de revisión conceptual.



Tenga en cuenta que los trabajos que produzca durante los procesos de estudio son insumos muy valiosos y de preparación para la Evaluaciones Parciales. Por lo tanto, guarde sus notas, apuntes y gráficos, le serán de utilidad.